



FUNDAMENTOS DO THREE.JS COM **ROUND SIX**

YURI MEDEIROS

ÍNDICE

Aula 1: Introdução e organização	Página 1
Aula 2: Fundamentos do Three.js	Página 2
Aula 3: Carregando a Boneca	Página 5
Aula 4: Configurando a Boneca	Página 8
Aula 5: Cenário	Página 10
Aula 6: Player	Página 11
Aula 7: HUD	Página 13
Aula 8: Movimentos da Boneca	Página 14
Aula 9: Mecânica	Página 15

THREE.JS

AULA 1: INTRODUÇÃO E ORGANIZAÇÃO



Three.js é uma biblioteca JavaScript e interface de programação de aplicativo (API) usada para criar e exibir gráficos de computador em 3D animados, em um navegador da Web usando WebGL.

O código-fonte está hospedado em um repositório no GitHub gratuitamente.






Bibliotecas de alto nível como Three.js ou GLGE, SceneJS, PhiloGL ou uma série de outras bibliotecas tornam possível criar animações de computador 3D complexas que são exibidas no navegador sem o esforço necessário de um aplicativo autônomo tradicional ou um plug-in.

Em anexo com os materiais dessa aula você receberá o arquivo **three.min.js** com o código dessa biblioteca.

Além da biblioteca three.js vamos usar a biblioteca GLTFLoader, que será necessária para carregarmos o modelo 3D da boneca, no formato GLTF para o código.

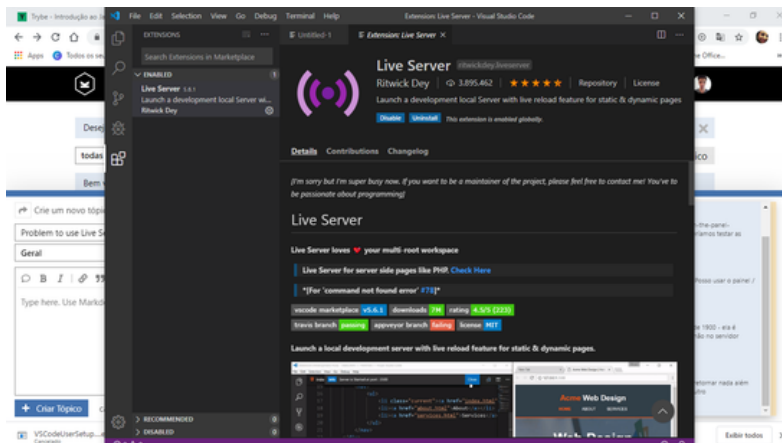
E por fim, usaremos a biblioteca GSAP, que criar animações de alta performance em poucos segundos, usaremos essa biblioteca para animar a nossa boneca.

Coloque os três arquivos que estão em anexo nos downloads dessa aula na pasta "bbt" e coloque a pasta bbt dentro de onde será a pasta do nosso jogo.

	GLTFLoader	24/10/2021 08:36	Arquivo JavaScript	102 KB
	gsap	24/10/2021 08:36	Arquivo JavaScript	63 KB
	three.min	24/10/2021 08:36	Arquivo JavaScript	600 KB

Começemos agora a organizar o projeto do nosso game, vamos programar usando o software VS CODE, em sua versão gratuita.

Além de instalar o plugin de Javascript, você precisará instalar um plugin chamado **Live Server**, responsável por hospedar dinamicamente os arquivos que vamos usar, para que o modelo 3D possa rodar normalmente no navegador.





No nosso arquivo .html vamos carregar as 3 bibliotecas e os futuros arquivos game.js e style.css que criaremos em seguida:

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8">
    <title>Batatinha Frita 1, 2, 3..</title>
    <style>
      body { margin: 0; }
    </style>
  </head>
  <body>
    <script src="bbt/three.min.js"></script>
    <script src="bbt/GLTFLoader.js"></script>
    <script src="bbt/gsap.js"></script>
    <script src="game.js"></script>
  </body>
</html>
```

Em seguida, crie os arquivos em branco:

- game.js
- style.css

E sua pasta ficará assim:

 bbt	08/12/2021 10:23	Pasta de arquivos	
 game	08/12/2021 10:47	Arquivo JavaScript	0 KB
 index	08/12/2021 10:46	Chrome HTML Do...	1 KB
 style	08/12/2021 10:47	Documento de fol...	0 KB

Abra o VS CODE, e abra a pasta com os arquivos em: File > OpenFolder > jogobatatinhafrita.

A seguir, aprenderemos os fundamentos do THREE.JS.

AULA 2: FUNDAMENTOS DO THREE.JS



O objetivo desta seção é fornecer uma introdução ao three.js. Começaremos montando uma cena, com um cubo giratório.

Para sermos capazes de exibir qualquer coisa com three.js, precisamos de três coisas:

- Cena,
- Câmera,
- Renderizador.

Para que possamos renderizar a cena com a câmera.

Em game.js escreva o código:

```
// Configura uma nova cena
const scene = new THREE.Scene();
// Câmera do tipo "Perspective" com campo de visão limitado conforme parâmetros
const camera = new THREE.PerspectiveCamera( 75, window.innerWidth / window.innerHeight, 0.1, 1000 );
// Declara um renderizador
const renderer = new THREE.WebGLRenderer();
// Tamanho da tela
renderer.setSize( window.innerWidth, window.innerHeight );
// Linka o renderizador no arquivo index.html
document.body.appendChild( renderer.domElement );
```

Configuramos então a cena, nossa câmera e o renderizador.

Existem algumas câmeras diferentes em three.js. Por enquanto, vamos usar uma

PerspectiveCamera, com o seguinte construtor:

PerspectiveCamera(fov : Number, aspect : Number, near : Number, far : Number).

O primeiro atributo é o campo de visão. **FOV** é a extensão da cena que é vista na tela a qualquer momento. O valor está em graus.

O segundo é a **proporção da imagem**. Quase sempre você deseja usar a largura do elemento dividida pela altura ou obterá o mesmo resultado de quando reproduz filmes antigos em uma TV widescreen – a imagem parece comprimida.

Os próximos dois atributos são os **planos de recorte próximos e distantes**. O que isso significa é que os objetos mais distantes da câmera do que o valor de longe ou mais perto do que perto não serão renderizados. Você não precisa se preocupar com isso agora, mas pode querer usar outros valores em seus aplicativos para obter um melhor desempenho.

O próximo é o **renderizador**. É aqui que a mágica acontece. Além do WebGLRenderer que usamos aqui, three.js vem com alguns outros, geralmente usados como substitutos para usuários com navegadores mais antigos ou para aqueles que não têm suporte para WebGL por algum motivo.

Além de criar a instância do renderizador, também precisamos definir o tamanho no qual queremos renderizar nosso aplicativo. É uma boa ideia usar a largura e a altura da área que queremos preencher com nosso aplicativo, neste caso, a largura e a altura da janela do navegador. Para aplicativos de **alto** desempenho, você também pode fornecer valores menores a **setSize**, como **window.innerWidth / 2** e **window.innerHeight / 2**, o que fará com que o aplicativo seja renderizado em um quarto do tamanho.

E por último, mas não menos importante, adicionamos o elemento renderizador ao nosso documento **HTML**. Este é um elemento <canvas> que o renderizador usa para mostrar a cena para nós.

Agora vamos instanciar o cubo:

```
// Declara o novo cubo
const geometry = new THREE.BoxGeometry();
// Declara um material para o cubo
const material = new THREE.MeshBasicMaterial( { color: 0x00ff00 } );
// Aplica o material no cubo
const cube = new THREE.Mesh( geometry, material );
// Adiciona o cubo a cena
scene.add( cube );
// Configura a profundidade da câmera
camera.position.z = 5;
```

Para criar um cubo, precisamos de um **BoxGeometry**. Este é um objeto que contém todos os pontos (vértices) e preenchimento (faces) do cubo, existem outras formas geométricas possíveis.

Além da geometria, precisamos de um **material** para colorir. Three.js vem com vários materiais, mas vamos nos limitar ao **MeshBasicMaterial** por enquanto. Todos os materiais possuem um objeto de propriedades que serão aplicadas a eles. Para manter as coisas muito simples, fornecemos apenas um atributo de cor de 0x00ff00 , que é verde. Isso funciona da mesma maneira que as cores funcionam em CSS ou Photoshop (cores hexadecimais).

A terceira coisa de que precisamos é uma **malha**. Uma malha é um objeto que pega uma geometria e aplica um material a ela, que podemos inserir em nossa cena e nos mover livremente.

Por padrão, quando chamamos scene.add () , o que adicionamos será adicionado às coordenadas (0,0,0) . Isso faria com que a câmera e o cubo ficassem um dentro do outro. Para evitar isso, simplesmente movemos um pouco a câmera.

Se executarmos o código agora, ainda não será possível ver o cubo.

Isso ocorre porque não estamos realmente renderizando nada ainda. Para isso, precisamos do que chamamos de **loop de renderização ou animação**.

```
function animate() {
  requestAnimationFrame( animate );
  renderer.render( scene, camera );
}
animate();
```

Essa função criará um loop que faz com que o renderizador desene a cena toda vez que a tela for atualizada (em uma tela normal, isso significa 60 vezes por segundo). Se você é novo na criação de jogos no navegador, pode dizer "por que não criamos apenas um setInterval?" A questão é, poderíamos, mas requestAnimationFrame tem uma série de vantagens. Talvez o mais importante seja que ela pausa quando o usuário navega para outra guia do navegador, portanto, não desperdiçando seu precioso poder de processamento e vida útil da bateria.

Agora sim, poderemos ver o cubo.

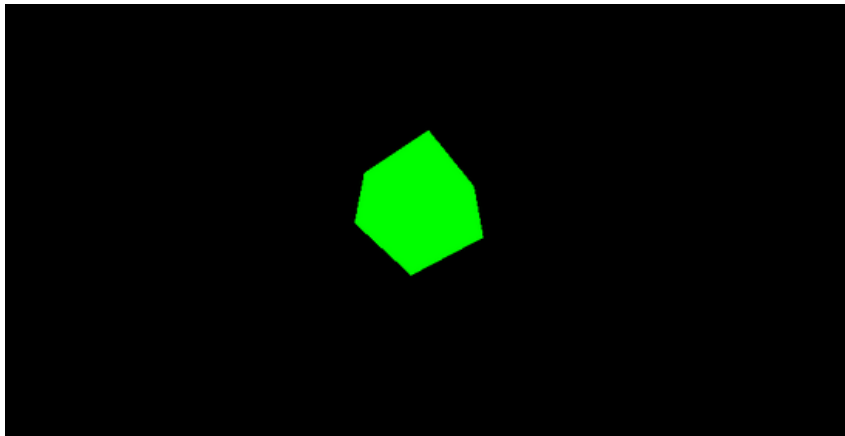
Vamos tornar tudo um pouco mais interessante girando o cubo.

Adicione o seguinte logo acima da chamada `renderer.render` em sua função `animate` :

```
function animate() {  
  requestAnimationFrame( animate );  
  cube.rotation.x += 0.01;  
  cube.rotation.y += 0.01;  
  renderer.render( scene, camera );  
}  
animate();
```

Isso será executado a cada quadro (normalmente 60 vezes por segundo) e dará ao cubo uma bela animação de rotação. Basicamente, qualquer coisa que você deseja mover ou alterar enquanto o aplicativo está em execução deve passar pelo loop animado. Você pode, é claro, chamar outras funções a partir daí, de modo que não termine com uma função animada com centenas de linhas.

A seguir, vamos começar a criar o game batatinha frita 1, 2, 3.



AULA 3: CARREGANDO A BONECA



O cubo que construímos na aula passado ficou bem legal, mas é hora de trocá-lo por um modelo 3D mais complexo, com os materiais dessa aula você recebeu o modelo 3D da boneca em GLTF, e já carregamos a biblioteca GLTFLoader que possibilitará o link do código com o modelo 3D, então mãos a obra.

Coloque a pasta "model" do material da aula, na pasta com os arquivos do seu game. Remova as 4 linhas de declaração do cubo e as 2 linhas de rotação, e logo após a alteração da câmera instancie o loader:

```
// Instanciando o loader
const loader = new THREE.GLTFLoader();
// Carregando o modelo na pasta model
loader.load("../model/scene.glTF",function(glTF){
    scene.add(glTF.scene);
})
```

Tudo certo, execute e procure a boneca.

Confira se existe algum erro no console, caso não haja temos apenas um problema, está escuro.

Temos que adicionar uma luz nessa cena, vamos adicionar uma luz ambiente:

```
// Adicionando uma luz
const light = new THREE.AmbientLight( 0x404040 );
scene.add( light );
```

Conseguiremos então ver a boneca fora de escala com uma luz um pouco escura.



Nos parâmetros da luz, coloque seis vezes a letra F maiúscula, retirando o 404040, isso fará com que apliquemos uma luz branca à cena.

Agora vamos configurar a escala da boneca, reduzindo seu tamanho:

```
// Carregando o modelo na pasta model
loader.load("../model/scene.glTF",function(glTF){
    scene.add(glTF.scene);
    glTF.scene.scale.set(0.4,0.4,0.4);
})
```


Esse é o resultado até agora:



Agora vamos ajustar a sua posição:

```
// Carregando o modelo na pasta model
loader.load("../model/scene.gltf",function(gltf){
    scene.add(gltf.scene);
    gltf.scene.scale.set(0.4,0.4,0.4);
    gltf.scene.position.set(0,-1,0);
})
```

O modelo 3D ficará centralizado:



Essa centralização não permanece quando alteramos o tamanho da tela, vamos tornar esse layout responsivo: (15min)

Escreva o código ao final da chamada `animate()`:

```
// Captura alteração de resolução e chama a função
window.addEventListener( 'resize', onWindowResize, false )
// Função que torna a tela responsiva
function onWindowResize(){
    camera.aspect = window.innerWidth / window.innerHeight
    camera.updateProjectionMatrix()
    renderer.setSize( window.innerWidth, window.innerHeight )
}
```

E por fim vamos mudar a cor de fundo da tela, acima do ajuste da posição da câmera.

```
// Altera a cor de fundo
renderer.setClearColor(0x8601af,1);
```

Resultado até aqui:



AULA 4: CONFIGURANDO A BONECA



Uma classe é uma estrutura que abstrai um conjunto de objetos com características similares, ela define o comportamento de seus objetos através de métodos, e os estados possíveis destes objetos através de atributos.

Em outras palavras, uma classe descreve os serviços oferecidos por seus objetos e quais informações eles podem armazenar.

Classes não são diretamente suportadas em todas as linguagens, e são necessárias para que uma linguagem seja orientada a objetos.

Agora, vamos transformar a nossa boneca em um objeto, que poderá ser recriado quantas vezes forem necessárias, mas para isso precisaremos que a boneca se transforme em uma classe, vamos começar declarando a classe boneca e um construtor, que contém o carregamento do modelo 3D, também vamos declarar uma variável que receberá a classe boneca, construa o código:

```
class boneca{
  constructor(){
    // Carregando o modelo na pasta model
    loader.load("../model/scene.glTF",function(glTF){
      scene.add(glTF.scene);
      glTF.scene.scale.set(0.4,0.4,0.4);
      glTF.scene.position.set(0,-1,0);
      this.Boneca1 = glTF.scene;
    })
  }
}

let Boneca1 = new boneca();
```

Agora nossa boneca começará a ganhar vida, através das funções "praTras" e praFrente" ela será capaz de executar um movimento de rotação, com o auxílio da biblioteca GSAP. Começamos alterando o tipo de função do loader, para uma arrow, seta:

```
class boneca{
  constructor(){
    // Carregando o modelo na pasta model
    loader.load("../model/scene.glTF",(glTF)=>{
      scene.add(glTF.scene);
      glTF.scene.scale.set(0.4,0.4,0.4);
      glTF.scene.position.set(0,-1,0);
      this.Boneca1 = glTF.scene;
    })
  }
}
```

Na sequência do construtor e ainda na classe boneca, crie as funções:

```
praTras(){
  gsap.to(this.Boneca1.rotation, {y:-3.15,duration:1});
}

praFrente(){
  gsap.to(this.Boneca1.rotation, {y:0, duration:1});
}
```

Logo mais vamos configurar para que os movimentos da boneca sejam feitos em intervalos de tempo específicos, por enquanto isso vamos declarar para que ela volte a olhar para trás a cada 1.000 milissegundos:

```
let Boneca1 = new boneca();  
setTimeout(()=> {  
  Boneca1.praTras},1000);
```

Execute as funções no console do navegador para ver os movimentos.

```
> Boneca1.praTras()  
< undefined  
> Boneca1.praFrente()  
< undefined
```

AULA 5: CENÁRIO



Agora vamos criar a árvore em que a boneca vai fazer a famosa contagem até três, vamos adicionar uma árvore lowpoly para economizar processamento.

Coloque a pasta "tree" na pasta do seu game e escreva o trecho:

```
loader.load("../tree/scene.glTF",function(glTF){
    scene.add(glTF.scene);
    glTF.scene.scale.set(16,16,16);
    glTF.scene.position.set(0,-6,-12);

    })

class boneca{
```

E nosso cenário ficará assim:



AULA 6: PLAYER



Logo acima da classe boneca, crie a classe player para configurarmos o nosso player carregando o emoji que vimos no início da aula:

```
class Player {
  constructor(){
    const geometry = new THREE.BoxGeometry( 0.3, 0.3, 0.3 );
    const material = new THREE.MeshBasicMaterial( {color: 0xffffff} );
    const player = new THREE.Mesh( geometry, material );
    scene.add( player );
    this.player = player;

    player.position.x = 3
    player.position.z = 0
    player.position.y = 0
  }
}
```

Na sequência vamos criar o PlayerInfo para melhor organizar as propriedades do player, ainda dentro do construtor:

```
this.playerInfo={
  positionX: 6,
  velocity: 0
}
```

A variável positionX controla a posição inicial do player e variável velocity controla a sua velocidade que começa em zero.

Crie uma instância de Player:

```
let Player1 = new Player();
```

Na sequência vamos criar o método anda() que colocará em 0.1 a velocidade do player sempre que for chamado:

```
anda(){
  this.playerInfo.velocity = 0.1;
}
```

Logo depois vamos criar o método update() que move o player:

```
update(){
  this.playerInfo.positionX -= this.playerInfo.velocity
  this.player.position.x = this.playerInfo.positionX
}
```

Crie a função para() para parar o player:

```
para(){
  this.playerInfo.velocity = 0;
}
```

Atualize a função `animate()` para chamar o `update()` em 60fps:

```
function animate() {  
    requestAnimationFrame(animate);  
    renderer.render( scene, camera );  
    Player1.update();  
}
```

Agora vamos configurar o `addEventListener`, que vai chamar a função `anda()` quando pressionarmos a seta para a esquerda (codigo 37) e quando o botão é solto o player para, dado que a função `para()` é chamada:

```
// pressiona a tecla  
window.addEventListener('keydown',function(e){  
    if(e.keyCode == 37){  
        Player1.anda()  
    }  
})  
// libera a tecla  
window.addEventListener('keyup',function(e){  
    if(e.keyCode == 37){  
        Player1.para()  
    }  
})
```

AULA 7: HUD



Agora vamos criar o placar com os textos que vão orientar a lógica do nosso game, começaremos no arquivo index.html criando uma tag de parágrafo.

```
<body>
  <p class="text">Esperando..</p>
  <script src="bbt/three.min.js"></script>
  <script src="bbt/GLTFLoader.js"></script>
  <script src="bbt/gsap.js"></script>
  <script src="game.js"></script>
</body>
```

Carregue o arquivo style.css no arquivo index.html:

```
<head>
  <meta charset="utf-8">
  <title>Batatinha Frita 1, 2, 3..</title>
  <link rel="stylesheet" type="text/css" href="style.css"></link>
  <style>
    body { margin: 0; }

  </style>
</head>
```

Nosso arquivo já possuirá as configurações básicas de margem e da classe "text":

```
{
  margin: 0;
  padding: 0;
}

.text{
  position: fixed;
  color: white;
  font-size: 30px;
  font-family: Arial, Helvetica,sans-serif;
  font-weight: bold;
  left: 50%;
  top: 10px;
  text-shadow: 0 2px 10px rgba(0,0,0,0.3);
  transform: translateX(-50%);
}
```


AULA 8: MOVIMENTOS DA BONECA



A seguir vamos programar as funcionalidades e sincronizar os movimentos da boneca, começaremos criando um método chamado "start" que define que a boneca começará o game de costas para o player:

```
start(){  
  this.praTras();  
}
```

A boneca terá um movimento periódico, virando de trás pra frente e vice versa, vamos criar uma função delay que vai separar esse intervalo de tempo entre a virada pra tras de uma nova virada para frente, crie acima da declaração da classe boneca:

```
function delay(ms){  
  return new Promise(resolve => setTimeout(resolve,ms));  
}
```

Retorne a função start() e coloque mais trechos para que a boneca realize os movimentos de virar pra tras e depois retornar a frente: (O await pausa a função até que o Promise esteja resolvido, aqui ele pausa por 1000 ms.

```
async start(){  
  this.praTras();  
  await delay(1000);  
  this.praFrente();  
  await delay(1000);  
  this.start();  
}
```

Use o console do navegador para chamar a função start da Boneca1:

```
> Boneca1.start()
```

A boneca começara a fazer movimentos periódicos.

Agora, vamos deixar esses movimentos aleatórios, para que o jogador não consiga calcular exatamente quando a boneca vai virar:

```
async start(){  
  this.praTras();  
  await delay((Math.random()*1000)+1000);  
  this.praFrente();  
  await delay((Math.random()*1000)+1000);  
  this.start();  
}
```

AULA 9: MECÂNICA



Começaremos criando a função `init()`, que vai configurar o início do jogo de forma automática, mas antes vamos declarar uma constante que fará referência ao texto que vamos atualizar de acordo com os estados do jogo:

```
const text = document.querySelector(".text")
```

Agora vamos criar a função `init()` que controlará a contagem regressiva para iniciarmos o jogo:

```
async function init(){
  await delay(500);
  text.innerText = "Começando em 3"
  await delay(500);
  text.innerText = "Começando em 2"
  await delay(500);
  text.innerText = "Começando em 1"
  await delay(500);
  text.innerText = "VAI!"
  startGame()
}
```

Em seguida criaremos a função `startGame()` que começa o jogo e colocaremos uma chamada da função `init()`:

```
function startGame(){
  Boneca1.start()
}
init()
```

Agora vamos criar uma nova constante, que define o tempo máximo de uma sessão do jogo, que será de 10 segundos.

```
const tmaximo = 10
```

Em seguida vamos criar a variável `gamestatus` que controla os diferentes estados de jogo:

```
let gamestatus = "esperando"
```

Um dos estados será "jogando", se não estivermos nele não poderemos mover o player:

```
window.addEventListener('keydown',function(e){
  if(gamestatus !== "jogando") return
  if(e.keyCode == 37){
    Player1.anda()
  }
})
```

Esse status será atribuído quando o jogo é iniciado:

```
function startGame(){  
  gamestatus = "jogando"  
  Boneca1.start()  
}
```

Antes de criarmos uma função para checar se o player está movendo quando a boneca está virada, vamos criar uma variável booleana que nos dirá se a boneca está ou não virada para trás ou não.

```
let tadecostas = true
```

Agora vamos atualizar o valor dessa variável conforme o giro da boneca, repare que o tempo é menor que ela está ficando de frente, dado que ainda poderemos nos mover um pouco quando a boneca começa a ficar de frente:

```
praTras(){  
  gsap.to(this.Boneca1.rotation, {y:-3.15,duration:1});  
  setTimeout(()=> tadecostas=true,450)  
}  
  
praFrente(){  
  gsap.to(this.Boneca1.rotation, {y:0, duration:1});  
  setTimeout(()=> tadecostas=false,150)  
}
```

Na classe Player, crie um novo método chamado "checa" e chame esse método dentro do método update.

```
checa(){  
}  
  
update(){  
  this.checa()  
  this.playerInfo.positionX -= this.playerInfo.velocity  
  this.player.position.x = this.playerInfo.positionX  
}
```

O método checa vai chegar se o player possui alguma velocidade quando a boneca está virada para frente, ou seja, não está de costas.

```
checa(){  
  if(this.playerInfo.velocity>0 && !tadecostas){  
    alert("perdeu")  
  }  
}
```

Agora vamos criar a condição que determina quando o player ele alcança a posição -6 do lado esquerdo vencendo o jogo:

```
if(this.playerInfo.positionX < -6){  
    alert("venceu")  
}
```

Execute e veja ambas as condicionais funcionando.

Agora vamos atualizar a condição de fim de jogo, colocando o valor "fimdejogo" na variável gameStatus, para que consigamos tratá-la adiante, indicando que o jogo acabou:

```
checa(){  
    if(this.playerInfo.velocity>0 && !tadecostas){  
        alert("perdeu")  
        gamestatus = "fimdejogo"  
    }  
  
    if(this.playerInfo.positionX < -6){  
        alert("venceu")  
        gamestatus = "fimdejogo"  
    }  
}
```

Na função animate(), vamos tratar essa condição:

```
function animate() {  
    if(gamestatus == "fimdejogo") return  
    requestAnimationFrame(animate);  
    renderer.render( scene, camera );  
    Player1.update();  
}
```

Em seguida, em vez de exibir um alerta com o status da partida, vamos atualizar o placar do jogo para "Você venceu!" ou "Você perdeu".

```
checa(){  
    if(this.playerInfo.velocity>0 && !tadecostas){  
        text.innerText = "Você perdeu"  
        gamestatus = "fimdejogo"  
    }  
  
    if(this.playerInfo.positionX < -6){  
        text.innerText = "Você venceu!"  
        gamestatus = "fimdejogo"  
    }  
}
```

E por fim, vamos configurar o terceiro final do jogo, por timeout.

```
function startGame(){
  gamestatus = "jogando"
  Boneca1.start()
  setTimeout(()=>{
    if(gamestatus != "fimdejogo"){
      text.innerText = "Timeout"
      gamestatus = "fimdejogo"
    }
  }, tmaximo*1000)
}
```